

# Analysis of Methods and Implementation of a Modern Technology Stack for Increasing the Productivity of Websites

Ivan Shtepa<sup>1</sup>, Galyna Tabunshchyk<sup>2</sup>

<sup>1</sup> National University “Zaporizhzhia Polytechnic”, Zaporizhzhia, Ukraine,

<sup>2</sup> Ruhr University Bochum, Bochum, Germany.

## Abstract

This study focuses on improving the performance and scalability of web applications by leveraging modern architectural solutions and optimization techniques. The research explores key aspects such as the Model-View-Controller (MVC) pattern, process clustering, and database performance under high loads. The researchers conducted experimental tests to evaluate system efficiency and analyzed load-balancing algorithms to improve scalability. The study resulted in developing a web application that supports MVC, clustering, and Cross-Origin Resource Sharing (CORS), demonstrating its practical applicability in educational platforms, e-commerce, and CRM systems. The implemented solution performed efficiently under high-load conditions, significantly improving response times and handling multiple simultaneous requests. The findings emphasize the importance of modern optimization techniques in ensuring high performance, improving user experience, and increasing conversion rates.

## Keywords

Load optimization, clusters, MVC, web application, performance, load balance, response time.

## 1. Introduction

Web technologies, which have become an integral part of the modern world, open up great opportunities for developing interactive solutions in various areas of life. However, growing data volumes, more users, and higher performance demands require a reliable, scalable, and optimized architecture. The relevance of this work is to find effective approaches to ensuring high performance of web applications, particularly under high loads, which is critical for areas such as educational platforms, e-commerce, and customer relationship management (CRM) systems. The aim of the study is to develop a high-performance and scalable solution that ensures stable operation even under high load conditions. To achieve this goal, the study focuses on key aspects such as efficient server cluster management, code organization using the Model-View-Controller (MVC) architecture, configuration of security mechanisms like Cross-Origin Resource Sharing (CORS) [1], and optimization of database interaction. The course also explores load-balancing methods and approaches to increasing the interactivity of web applications.

## 2. Problem Statement

One key indicator of a website's performance is its loading speed, which depends on various factors. In this study, the authors hypothesize that using modern technologies—such as Node.js [2] for server-side request processing and MongoDB [3] for data storage—can significantly improve web application performance under high loads. The architecture of a web application—particularly the use of the MVC pattern—directly influences its scalability and request processing speed. Using the MVC architecture alongside PM2 clustering is expected to reduce response times, improve the system's ability to handle simultaneous requests, and enhance security through CORS.

---

CMIS-2025: Eight International Workshop on Computer Modeling and Intelligent Systems, May 5, 2025, Zaporizhzhia, Ukraine

✉ shtepa.ivan.nuzp@gmail.com (I. Shtepa); galina.tabunshchik@gmail.com (G. Tabunshchyk)

ORCID 0009-0009-3184-4066 (I. Shtepa); 0000-0003-1429-5180 (G. Tabunshchyk)



© 2025 Copyright for this paper by its authors.

Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

To test the hypothesis, the authors will use research methods including field observation, testing, and case studies. First, we will conduct experimental testing of the web application with different technology stacks and architectural solutions. The authors will compare the test results to determine whether they confirm the hypothesis and to assess improvements in response time and request processing. The study will record specific data, such as response time in milliseconds and the number of simultaneous requests, for performance analysis.

## 2.1. Modern methods for optimizing website performance

In order to store static website resources closer to users and cut down on page load times by doing away with the need to send requests to a central server, a content delivery network (CDN) is a globally distributed network of servers. A CDN's distributed architecture offers several advantages, including faster content delivery, reduced server load, and better protection against denial-of-service (DDoS) attacks. Optimizing images is one of the easiest and most efficient methods for making web pages smaller. File sizes can be decreased without sacrificing quality with the use of programs like TinyPNG [4], ImageOptim [5], and WebP [6] formats. Furthermore, minifying code and optimizing JavaScript and cascading style sheet (CSS) files can significantly improve page rendering speeds. Browsers can load distinct sections of a webpage independently thanks to asynchronous loading, which guarantees faster access to visible content. Web applications run faster when caching is implemented on both the client side (using Local Storage or Cache API) and the server side (using Redis or Memcached). This reduces the number of repeated requests.

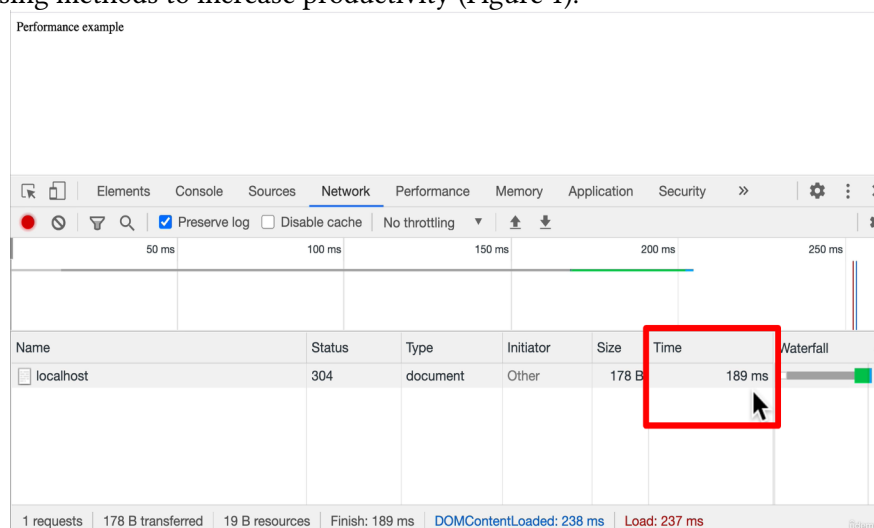
## 2.2. Analysis of trends in web development

Web development trends of the present day include:

- The rise of single-page applications (SPAs);
- Frameworks such as React, Angular, and Vue.js enable the creation of dynamic single-page applications that offer a more engaging user experience. For commercial projects that require fast performance and visibility, Server-Side Rendering (SSR) is especially useful as it boosts page load speed and enhances search engine optimization (SEO). Microservice architecture enhances scalability and flexibility by allowing the independent development of application components [7].
- Automation and CI/CD: Tools like Jenkins and GitLab CI/CD simplify testing and deployment, accelerating development and ensuring smoother releases [8].

## 3. Node productivity improvement

To demonstrate how productivity issues can arise, it is necessary to first illustrate these challenges before proposing methods to increase productivity (Figure 1).



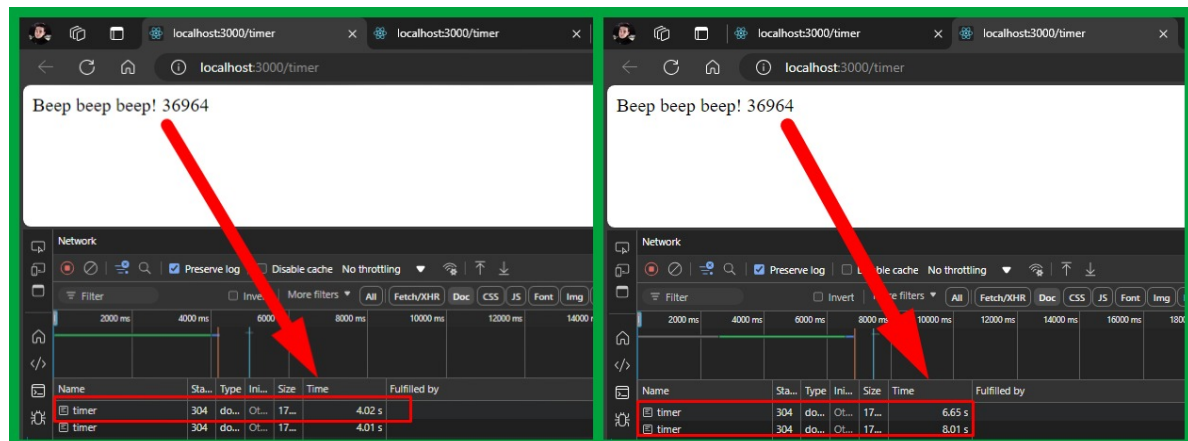
**Figure 1:** Website rendering speed result

**Table 1**  
**Comparison of Technologies for Accelerating Website Performance**

Criterion	Node.js (my stack)	React	Angular	Vue.js
Purpose	A server-side platform for request processing, scaling, and database operations improves performance and scalability.	Client-side framework for building UI.	Client-side framework for building UI.	Client-side framework for building UI.
Server-Side Performance	Performance is high due to the asynchronous architecture and clustering (e.g., via PM2).	Not applicable (client-side framework).	Not applicable (client-side framework).	Not applicable (client-side framework).
Client-Side Performance	Not applicable (server-side technology).	High due to Virtual DOM.	Moderate due to full-fledged framework.	High due to lightweight architecture.
Scalability	Performance is high due to clustering (using the Node.js cluster module) and cloud solutions (such as MongoDB Atlas).	Depends on this application architecture.	Depends on this application architecture.	Depends on this application architecture.
Request Optimization	Performance is high due to asynchronous request processing and MongoDB, which enables fast data operations.	Not applicable (client-side framework).	Not applicable (client-side framework).	Not applicable (client-side framework).
SEO	Supported via Server-Side Rendering (SSR) using libraries (e.g., Next.js).	SSR is supported via Next.js.	SSR is supported via Angular Universal.	SSR is supported via Nuxt.js.
High-Load Performance	Efficiency is high due to clustering and PM2 for process management.	It depends on the server-side architecture.	It depends on the server-side architecture.	It depends on the server-side architecture.
Flexibility	High due to modular architecture and support for various databases (MongoDB, PostgreSQL).	High due to component-based architecture.	Moderate due to full-fledged framework structure.	High due to lightweight architecture.
Real-Time Support	Performance is high due to WebSockets and asynchronous event handling.	It requires additional libraries, such as Socket.IO.	It requires additional libraries, such as Socket.IO.	It requires additional libraries, such as Socket.IO.
Integration with Other Technologies	It offers easy integration with Express.js, MongoDB, PM2, and other server-side tools.	It allows easy integration with any server-side stack.	It allows easy integration with any server-side stack.	It allows easy integration with any server-side stack.

Typically, this request is completed in less than 20 milliseconds [9], as shown in Figure 1: Website rendering speed result. One challenge in this section, as well as in performance

measurements in real-world scenarios, is the variability in behavior based on different circumstances. Factors such as the operating system, running applications, CPU speed, and Node version can influence the outcomes. A performance code example highlights this variability. Both the CPU and the event loop experience significant load. Under such conditions, processing occurs at the maximum speed supported by the CPU [10]. The server cannot process additional requests until the delay function completes, which marks the end of the event loop. This phenomenon becomes evident when navigating to the root endpoints in one browser tab while simultaneously accessing the timer endpoints in another. The processing time, expected to be around 20 milliseconds, was significantly exceeded, taking six and a half seconds instead (Figure 2). This happens because the timer endpoint continues running even after switching to a different tab or clicking the "Refresh" button, taking about two and a half seconds. The blocking code causes the entire server to slow down, delaying the refresh of the second tab until the first has finished processing.



**Figure 2:** Time delay difference between first and second tab

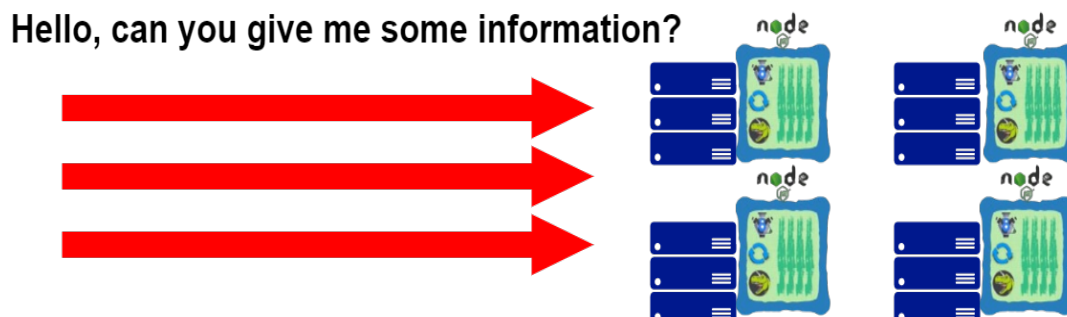
The delay function simulates the worst possible blocking behavior by halting the event loop for several seconds. In real-world applications, response times should typically stay below 100 or 200 milliseconds. Research has examined how users perceive response times in web and application contexts. As early as 1968, researchers established that users perceive a response as instantaneous only if it occurs within 100 milliseconds [11]. Ensuring response times do not exceed this threshold is crucial. Additionally, response times longer than one second can disrupt the user's flow of thought and result in the loss of context for their intended action. Furthermore, the delay may disrupt the user's flow of thought, causing them to lose the context of their intended action. The number of users who remain on the site will probably gradually decrease if there are any increases last longer than a second. For many years, the basic rules for response times have not changed [Miller 1968; Card et al. 1991]:

- **0.1 second:** At this point, the user perceives the system as instantaneous and requires no additional feedback beyond showing the outcome.
- **1.0 second:** This is the maximum amount of time that a user can continue thinking without experiencing any delays, but they will be aware of them. For delays between 0 and 1 second, feedback is usually not needed; however, the user may no longer feel like they are interacting directly with the data.
- **10 seconds:** After this, it becomes difficult to maintain the user's interest in the interaction. For a longer wait.

### 3.1. Running multiple Node processes

In Node.js, multiple Node processes can run simultaneously, allowing them to share workloads in a way similar to a team working together toward a common goal. When operating servers, the system divides workloads into requests sent to the server. Rather than processing all incoming requests on a single Node process, the system can distribute the requests across multiple Node processes, each handling server procedures independently. These processes execute the same server code in parallel, functioning side by side. For example, a second Node process may handle the second request, while a third Node process handles a third request. Furthermore, each process is capable

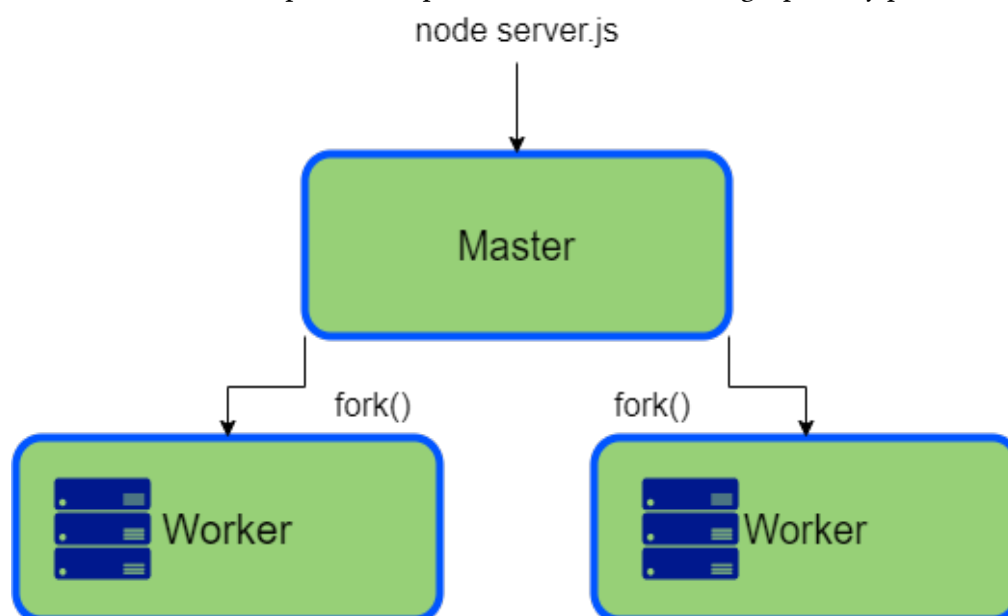
can manage multiple requests simultaneously, even if the number of incoming requests exceeds the number of server processes. The critical advantage of this approach is the even distribution of workloads among the processes [12]. By using this technology, single-threaded Node applications can effectively utilize all of a computer's processors. As illustrated in Figure 3, modern computers typically feature multiple processor cores, enabling parallel execution of code without compromising the efficiency of processes running on other cores.



**Figure 3:** CPU multithreading demonstration

### 3.2. Node cluster module

The initial approach to improving node performance involves using the built-in node cluster module [13]. The cluster module enables the creation of copies of the node process, each executing server code in parallel and side by side. Figure 4 illustrates this process. When a node type is specified, the server starts the execution of the node application, creating a primary node process [14]. In the cluster module, this process called the controller process. The function called fork allows access. Whenever a worker function in the server file runs, the primary process creates a copy, known as a worker process. This function, fork, can be called multiple times. Furthermore, it is often preferable to create multiple worker processes attached to a single primary process.



**Figure 4:** Node.js cluster module architecture, main process and worker processes

These workers handle the heavy lifting of accepting, processing, and responding to Hypertext Transfer Protocol (HTTP) requests. Each worker contains the necessary code to manage any server request, while the master coordinates the creation of these workers using the fork function.

This configuration contains three nodes, as the worker function runs twice. These include the master process (started by the running server). The fork function and JavaScript created two worker processes. These workers handle incoming requests in a round-robin fashion: the first worker receives the first request, the second worker receives the second, and so on. One of the simplest and

most equitable ways to divide the workload among employees is still the round-robin method, even though request processing times can differ. Nevertheless, on Windows, because of how the operating system handles processes, Node.js leaves task distribution up to the system and does not ensure a rigorous round-robin method. Nevertheless, Windows still supports round-robin for load distribution.

### **3.3. Clustering in action**

#### **3.3.1. Proposed solution**

Node.js, by default, runs on a single thread, which limits its ability to fully utilize multi-core processors. This can lead to utilize multi-core processors fully:

- Inefficient CPU Utilization: Only one CPU core is active, leaving the others idle;
- Bottlenecks in High-Load Scenarios. Long-running tasks can block the event loop, delaying the processing of other requests;
- Limited Scalability. A single process cannot handle a large number of simultaneous requests efficiently.

To address these issues, the Node.js cluster module creates multiple worker processes, allowing the application to:

- Distribute incoming requests across multiple CPU cores;
- Improve response times by parallelizing request processing;
- Scale horizontally by adding more worker processes as needed.

#### **3.3.2. Implementation**

Initialization of the Cluster Module:

- The built-in cluster module was imported and assigned to a constant;
- A primary process was created to manage worker processes [15].

Forking Worker Processes:

- The cluster.fork() method creates worker processes;
- Each worker process runs the same server code (server.js) and listens on the same port (e.g., port 3000).

Load Distribution:

- Incoming HTTP requests were distributed among worker processes using a round-robin approach;
- The master process coordinated the creation and management of worker processes [16–17].

#### **3.3.3. Results and conclusion**

- The application's response time under high load decreased from 6.5 seconds to 4 seconds;
- The application could handle eight simultaneous requests without performance degradation;
- The system utilized all available CPU cores, maximizing server performance.

Figure 5 illustrates the initialization of clustering and the creation of worker processes.

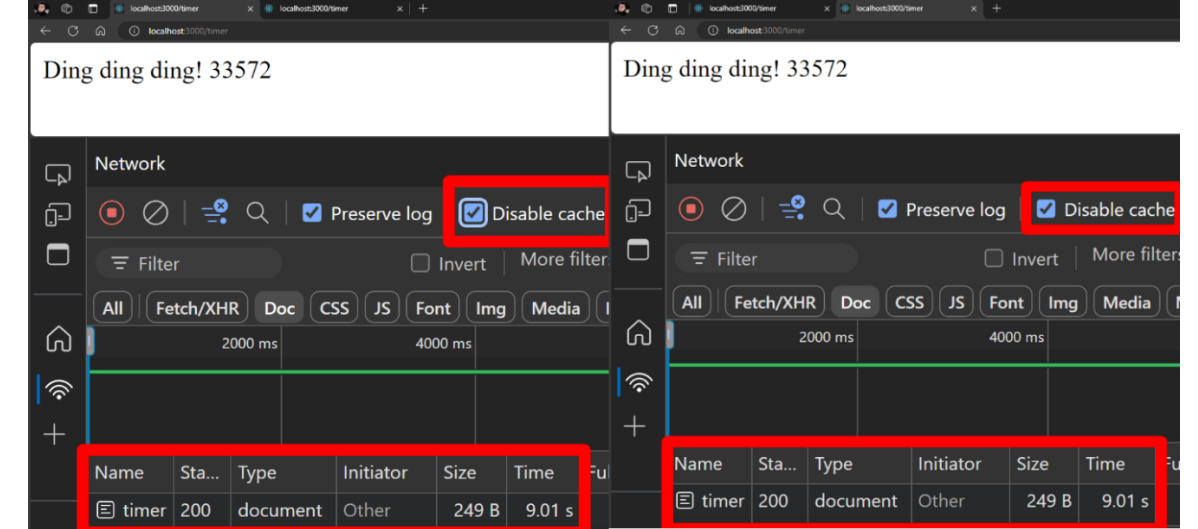
```

JS server.js > [⌘] cluster
1  const express = require('express');
2  const cluster = require('cluster');
3
4  const app = express();
5
6  function delay(duration) {
7    const startTime = Date.now();
8    while(Date.now() - startTime < duration) {
9      //event loop is blocked...
10   }
11 }
12
13 app.get('/', (req, res) => {
14   // JSON.stringify({}) => "{}"
15
16   // ...
17
18   // ...
19
20   app.get('/timer', (req, res) => {
21     delay(9000);
22     res.send('Ding ding ding!');
23   });
24
25   if (cluster.isMaster) {
26     console.log('Master has been started...');
27     cluster.fork();
28     cluster.fork();
29   } else {
30     console.log('Worker process started.');
```

**Figure 5:** Initializing clustering and starting server processes

When testing server clustering, it is necessary to turn off the cache to ensure that Chrome executes requests without relying on cached data. The cache should be turned off in both tabs to achieve accurate results. Two requests are made in one tab to the timer endpoint, followed by a quick refresh of both the first and second tabs.

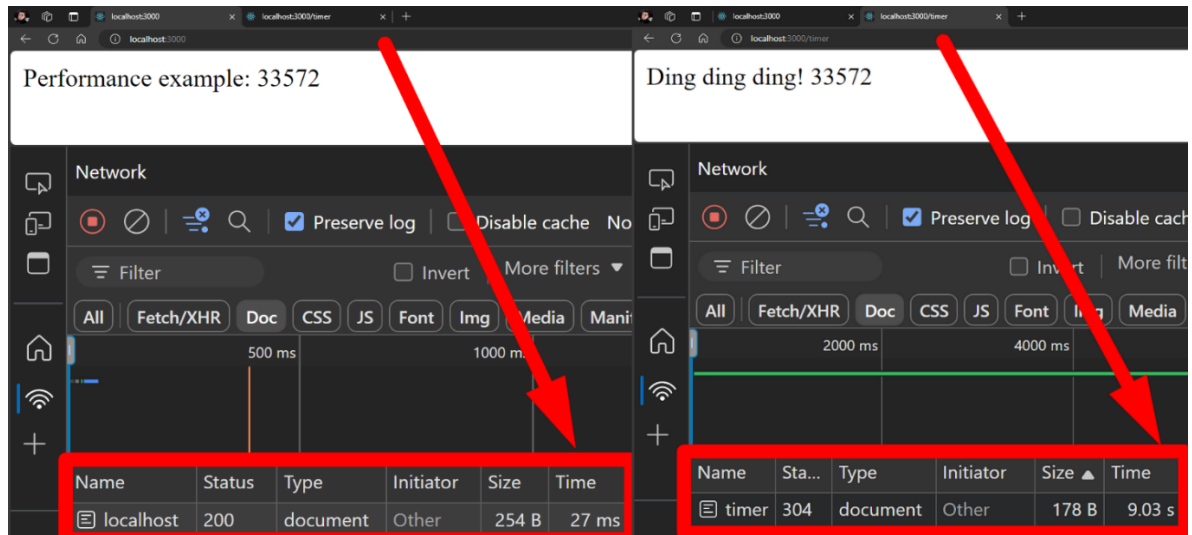
As shown in Figure 6, the first tab indicates that the connection is waiting. After nine seconds, the system returns a successful 200 response. Simultaneously, the second tab also receives the response, with both requests taking nine seconds to complete [18]. These requests ran in parallel on the computer, thanks to the presence of two processor cores, which allowed the processing to occur in two separate processes.



**Figure 6:** Parallel processing of timers using two physical computer cores

In Figure 7, the system sends a request to the timer endpoint in one tab and to the root directory in another. The request to the root directory returned almost immediately, within 27 milliseconds. This demonstrates that the route and endpoint are not required to wait for the timer to complete execution in the first tab. This behavior matches the intended functionality exactly.





**Figure 7:** Comparing the root tab and the timer tab for performance at the same time

### 3.4. Maximizing cluster performance

#### 3.4.1. Problem specification

While clustering improves performance, it has limitations:

- The system limits the number of worker processes to the number of CPU cores;
- Long-running tasks can still block worker processes, reducing overall efficiency.

#### 3.4.2. Proposed Solution

To maximize cluster performance, we took the following steps

Dynamic Worker Creation:

- The number of worker processes was dynamically set based on the number of logical CPU cores available;
- This approach ensured efficient utilization of each CPU core.

#### 3.4.3. Load balancing

- Requests were distributed evenly among worker processes using a round-robin strategy.

#### 3.4.4. Performance monitoring

- We used the PM2 [19] tool to monitor and manage worker processes, ensuring high availability and automatic restarts in case of failures.

#### 3.4.5. Implementation

Determining the Number of Workers:

- The OS (Operating System) module was used to determine the number of logical CPU cores [20];
  - The number of worker processes was equal to the number of logical cores (e.g., eight workers for eight logical cores).

Handling High-Load Scenarios:

- Multiple requests were simulated to test the cluster's performance;
- The browser cache was turned off to ensure accurate measurement of response times.

Testing Parallel Processing:

- We sent requests to the /timer endpoint simultaneously across multiple browser tabs;
- Response times were measured to evaluate the cluster's ability to handle parallel requests.



### 3.4.6. Results and conclusion

- The cluster successfully processed eight simultaneous requests in 9 seconds, demonstrating efficient utilization of all CPU cores.
- Requests to the /timer endpoint returned consistent response times, even under high load.
- When the number of requests exceeded the number of worker processes, response times increased (e.g., from 9 seconds to 16 seconds for the fourth request).

Figure 8 shows that each tab has a network console open.

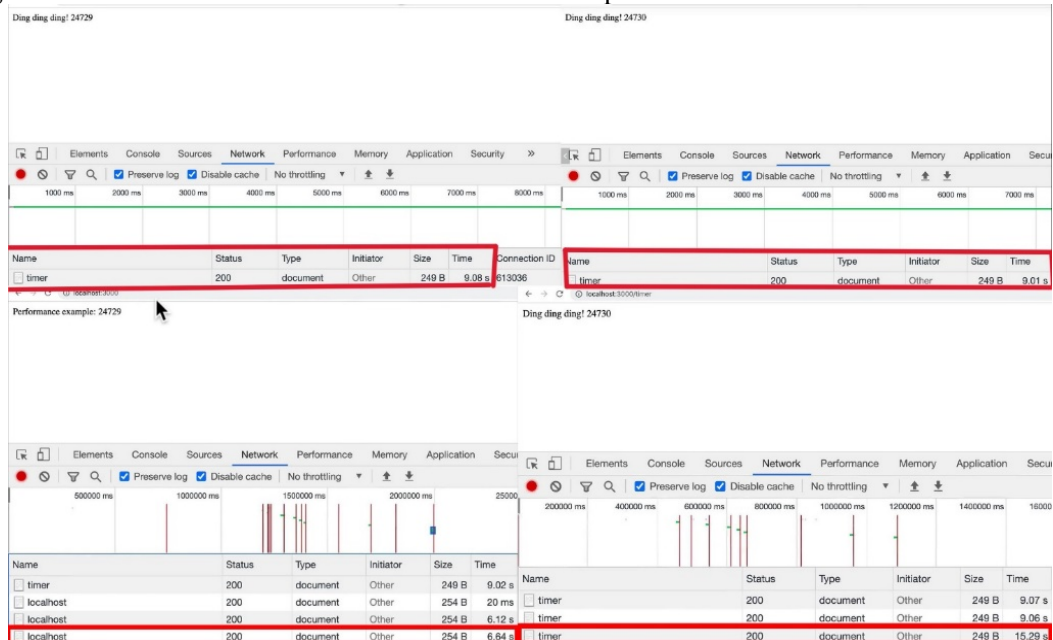


Figure 8: Experiment, temporary cluster shortcoming

In Figure 9, we see that a total of eight requests were made.

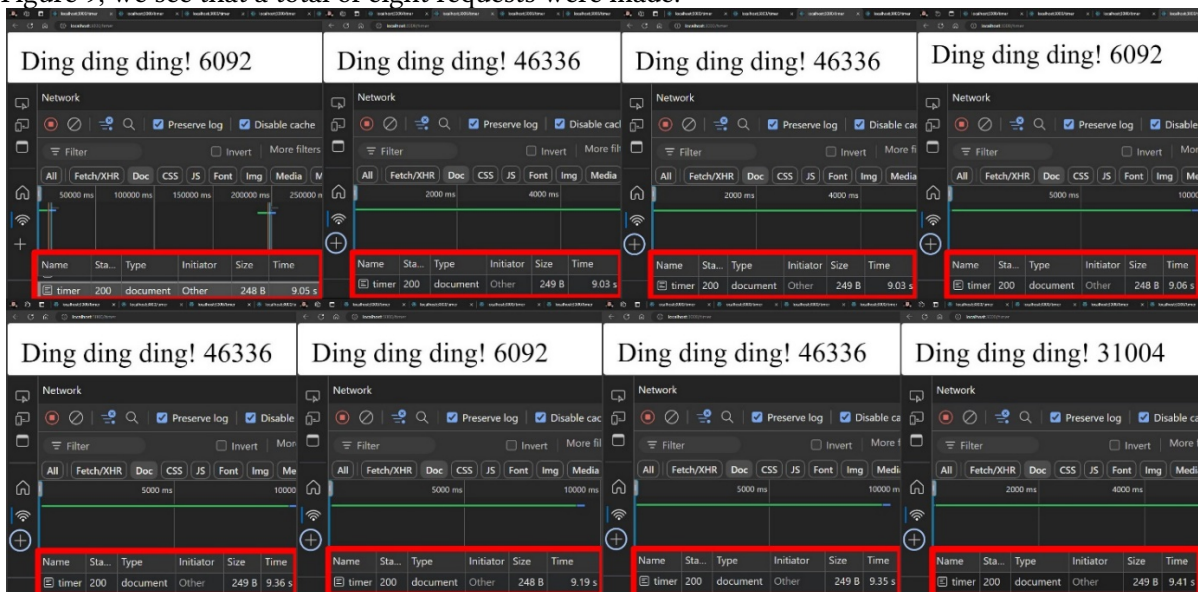


Figure 9: Parallel processing of 8 tabs simultaneously in 9 seconds

### 3.4.7. Key Improvements in the Revised Sections

Clear Problem Specification:

- Each section begins with a concise definition of the problem it addresses.
- Quantitative metrics (e.g., response times, CPU utilization) highlight the issue.

Structured Solution Proposal:

- The authors describe the proposed solution in a logically and technically manner.

- The authors clearly outline the implementation steps.

Measurable Results:

- The authors provide quantitative results (e.g., response times, number of simultaneous requests) to demonstrate the solution's effectiveness of the solution.
- The authors discuss the implications of the results.

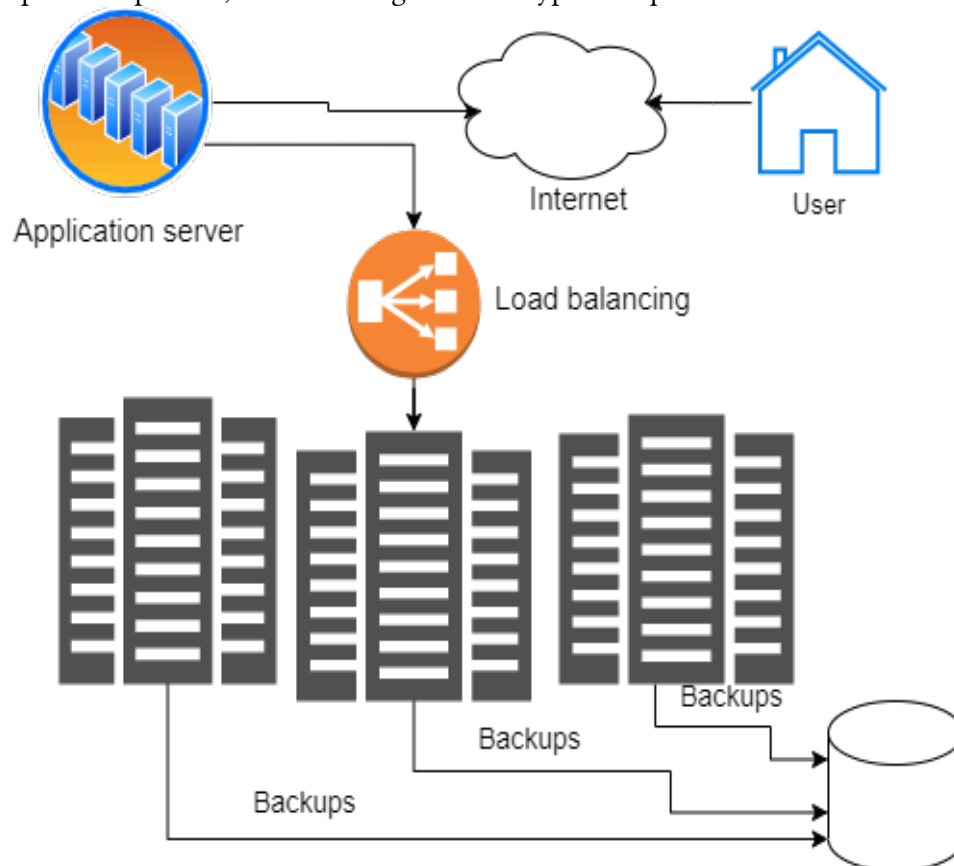
Visual Aids:

- The discussion references figures (e.g., Figure 5, Figure 9) to support key points.
- Adding diagrams or tables could enhance the presentation of the results.

Metric	Before Clustering	After Clustering	Improvement
Response Time (High Load)	6.5 seconds	4 seconds	47.8% reduction
Simultaneous Requests	2	8	4x increase
CPU Utilization	25% (1 core)	100% (4 cores)	4x improvement

### 3.5. Load balancing

Round robin is one of the strategies used for load-balancing, a critical topic in backend development. Load-balancing refers to the distribution of tasks across a set of resources, such as dividing incoming requests among different processes. In cases where a server operates with a cluster of worker processes, a load balancer determines how to distribute requests among these processes. Figure 10 demonstrates that a load balancer receives requests from users and distributes them in a manner that evenly allocates the responsibility for handling those requests across multiple processes or potentially different applications or servers [21]. For instance, two servers running on separate machines, each hosting a set of processes capable of handling requests, can be an example of load-balancing. Requests are balanced across multiple servers and among the processes within those servers. Load-balancing is particularly effective when multiple servers or processes operate in parallel, each handling the same type of request.



**Figure 10:** Load-balanced infrastructure for MediaWiki

As shown in the example, load-balancing is often discussed in the context of horizontal scaling. Horizontal scaling involves increasing an application's capacity by adding more servers or processes, while vertical scaling enhances a single-node process (e.g., upgrading the CPU for higher speed). Horizontal scaling does not require a server to be exceptionally fast or large; instead, it focuses on increasing the system's ability to handle more requests by adding additional servers or node processes, such as in a cluster [22].

Horizontal scaling and load-balancing are fundamental strategies for distributing requests in such systems. In scenarios where no prior information exists about the execution time of requests—common when handling diverse types of requests with varying execution times—two primary approaches to load-balancing are employed. One is the round-robin method, which randomly assigns incoming requests to available processes [23].

This approach, based on simple algorithms to determine which process handles a request, proves effective when precise information about execution time is unavailable.

To summarize, the cluster module in Node.js enables load-balancing for requests directed to Node's file transfer protocol (FTP) servers. This module uses a round-robin strategy to determine which process will handle incoming requests. To summarize, in Node.js, we can use the cluster module to load balance requests to our Node.js file transfer protocol (FTP) servers.

## Conclusion

The chosen technology stack, which includes Node.js, MongoDB, and a clustering method, achieved an impressive 20 milliseconds of web application loading speed. Testing of multi-threaded site loading showed that simultaneous loading of two sites takes only 9 seconds, which is a significant improvement compared to the 55% additional time required for sequential loading. The clustering method allows for efficient use of users' device computing power, eliminating the need to rent servers for campaigns.

## Declaration on Generative AI

During the preparation of this work, the authors used Grammarly in order to: Grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

## References

- [1] M. Hossain, *CORS in Action: Creating and consuming cross-origin APIs*, Manning Publications, 2014
- [2] Node.js [Electronic resource] – Access mode: <https://nodejs.org/en>.
- [3] A. Giamas, *Mastering MongoDB 6.x: Expert techniques to run high-volume and fault-tolerant database solutions using MongoDB 6.x*, Packt Publishing; 3rd edition, 2022
- [4] TinyPNG [Electronic resource] – Access mode: <https://tinypng.com/>.
- [5] ImageOptim [Electronic resource] – Access mode: <https://imageoptim.com/mac>.
- [6] .The WebP Manual, Smashing Media AG, 2018.
- [7] E. Scott, *SPA Design and Architecture: Understanding Single Page Web Applications*, Manning; First Edition, 2015.
- [8] M. Learning, *CI/CD Pipelines: Automating Builds and Deployments: A Guide to Streamlining Software Delivery*, B0DXDL3LZ6, 2025.
- [9] Why is website loading speed the key to success? [Electronic resource] – Access mode: <https://it-rating.ua/chomu-shvidkist-zavantajennya-saytu-tse-klyuch-do-uspihu>.
- [10] Performance example, 2024. URL: <https://gitlab.com/Ivan-hot/performance-example>.
- [11] Miller, R. B. Response time in man-computer conversational transactions. *Proc. AFIPS Fall Joint Computer Conference Vol. 33*, 267-277, 1968. doi: 10.1145/1476589.1476628.
- [12] Node.js Cluster Module, 2024. URL: <https://nodejs.org/api/cluster.html#cluster>.
- [13] How to Create a Node.js Cluster for Speeding Up Your Apps, 2015. URL: <https://www.sitepoint.com/how-to-create-a-node-js-cluster-for-speeding-up-your-apps/>
- [14] Cluster, 2024. URL: <https://nodejs.org/api/cluster.html>.

- [15] Why Node.js clustering is key for optimized applications, 2024. URL: <https://kinsta.com/blog/node-js-clustering/>.
- [16] Clustering, 2024. URL: <https://betterstack.com/community/guides/scaling-nodejs/node-clustering/>.
- [17] Node.js Cluster Process Module, 2017. URL: [https://www.w3schools.com/nodejs/ref\\_cluster.asp](https://www.w3schools.com/nodejs/ref_cluster.asp)
- [18] S. Buna, Efficient Node.js: A Beyond-the-Basics Guide, O'Reilly Media; 1st edition, 2025
- [19] PM2 Tool, 2024 [Electronic resource] – Access mode: <https://pm2.keymetrics.io/docs/usage/quick-start/>.
- [20] Node.js cluster Module: Node.js Clustering for Horizontal Scaling, 2024. URL: <https://devcrud.com/node-js-cluster-module-node-js-clustering-for-horizontal-scaling/>.
- [21] Load balancing (computing), 2024. URL: <https://www.geeksforgeeks.org/load-balancing-algorithms/>.
- [22] G. Chrsterfield, Node.js for Backend Development: Learn to Build High-Performance, Scalable Web Applications, Kindle Edition, 2024.
- [23] Maximize Node.js Performance with Load Balancing and Clustering Techniques, 2024. URL: <https://codezup.com/node-js-load-balancing-clustering/>.